ESTATECORE

The Core of Smart Real Estate

Submitted By

1. **Umar Ahmad-BSSE23032**
2. **Ali Hasnain - BSSE23066**

Submitted to

1. **Dr. Zunnurain Hussain**
2. **Sir Umair Makhdoom**

Table of Contents

Lists of Figures

Lists of Tables

Introductions

1.1 Background:

Large amounts of user traffic, multimedia content, and real-time search operations are all handled by current real estate platforms. These systems need scalable infrastructure, quick response times, and high availability as they expand. Due to their low fault tolerance and restricted scalability, traditional monolithic architectures, which were initially intended for small- scale systems, find it difficult to meet these needs.

Figure 1: Logo of EstateCore

1.2 Motivation

The workloads of real estate applications are often unequal, with agent operations involving fewer but crucial write transactions and consumer interactions generating high amounts of read requests. These workloads share resources in monolithic designs, which results in system failures and performance obstacles during updates. This challenge motivates the adoption of a cloud-native microservices architecture that supports independent scaling, fault isolation, and seamless continuous deployment.

1.3 Domain Relevance

The real-estate domain represents common challenges faced by modern web applications, including secure authentication, fast search capabilities, efficient media storage, and reliable data management. Designing a scalable and resilient solution for this domain demonstrates architectural principles that are directly applicable to other large-scale systems such as e-commerce platforms and Software-as-a-Service **(SaaS)** applications.

Problem Statement

2.1 Definition

Traditional real-estate platforms are frequently built using monolithic designs, in which customer facing features and agent-level administration processes are tightly linked into a single system. This coupling causes all user requests, including read-heavy browsing actions and write intensive administration procedures, to share the same infrastructure and resources.

2.2 Rationale

Such architectural coupling leads to multiple limitations, including poor scalability under high user traffic, system downtime during updates, limited fault isolation, and increased security risks. Read-heavy customer traffic can negatively impact critical agent operations, while failures in one component may affect the entire system. These limitations justify the need for a cloud-native, microservices-based solution that enables independent scaling, improved reliability, and uninterrupted system availability.

Stakeholder	Problem Faced	System Solution
Customers	Outdated property listings, slow search, lack of reliable information	Real-time property listings, fast search using OpenSearch, cached responses via Redis
Property Agents	Manual listing updates, delayed approvals, limited system control	Dedicated agent dashboard with secure create, update, and delete operations
Platform Administrator	Difficulty in monitoring system performance and user activity	Centralized data management, role-based access control, and monitoring via AWS services
System Owners (Business)	Poor scalability, downtime during updates, performance issues under load	Cloud-native microservices architecture with auto-scaling and zero-downtime deployments

Table 1: Stakeholders and Needs

Proposed Solution

EstateCore is proposed as a cloud-native real-estate platform based on a microservices architecture deployed on Amazon Web Services (AWS). The system is designed to address scalability, performance, and availability issues present in traditional monolithic real-estate applications. By separating customer-facing and agent-level functionalities into independent services, the platform supports high traffic loads, secure operations, and continuous deployment.

3.1 Customer Microservice (Read-Only Operations)

The Customer Microservice conducts all read-intensive actions such property browsing, searching, and filtering. It is optimized for large traffic with caching and rapid search methods, ensuring quick response times and a seamless user experience even during peak hours.

3.2 Agent Microservice (Read-Write Operations)

The Agent Microservice handles key write activities such as generating, updating, and removing property listings. This service is isolated for security and dependability, allowing critical processes to take place without interfering with customer-side performance.

Aspect	Existing System	EstateCore
Architecture	Monolithic	Microservices-based
Scalability	Limited, system-wide	Independent service scaling
Availability	Downtime during updates	Zero-downtime deployments
Performance	Degrades under load	Optimized with caching and CDN
Security	Basic controls	Multi-layer cloud security

Table 2: Estate vs Existing Systems

Features	Description
Property Browsing	Allows users to view and explore available properties with detailed information.
Advanced Search	Enables fast property search and filtering using optimized search mechanisms.
Agent Management	Allows agents to create, update, and delete property listings securely.
Performance Optimization	Improves response time through caching and content delivery networks.
Media Handling	Stores and serves property images efficiently using cloud storage.

Table 3: Features Details

Architecture Diagram

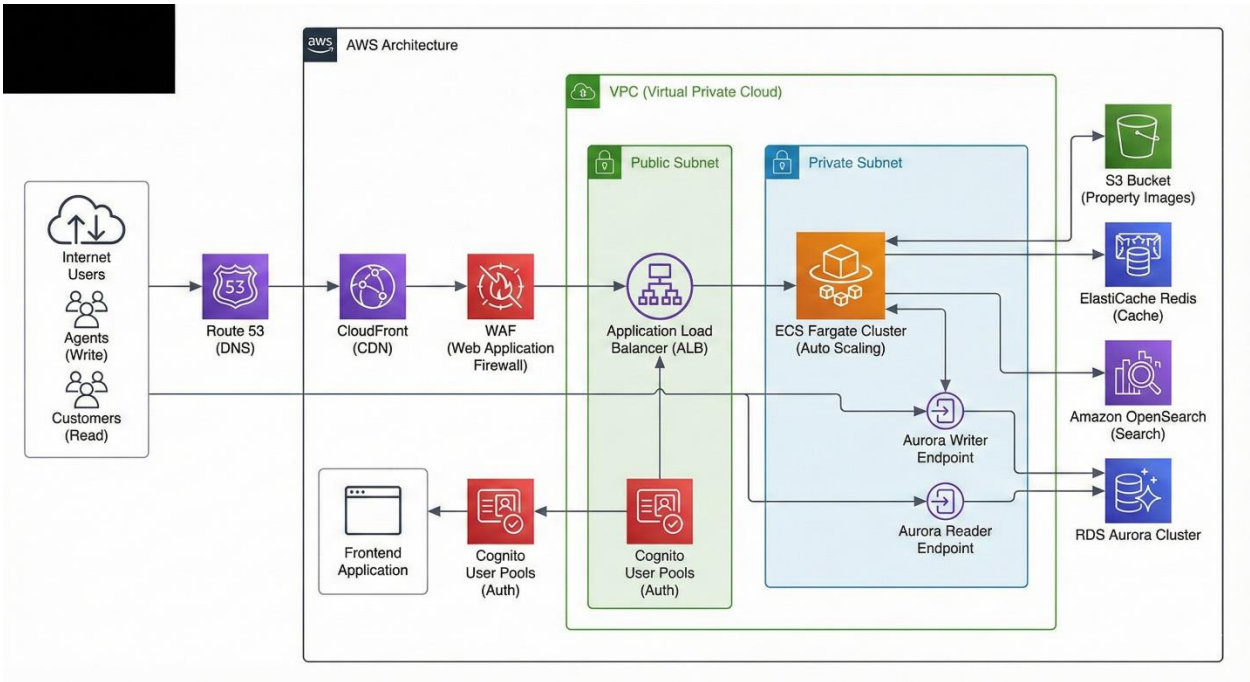

Figure 2: Architecture Diagram

4.1 Explanation

The system is implemented as a cloud-native microservices architecture on AWS. User requests are routed through Route 53, CloudFront, and WAF for optimized performance and security before reaching an Application Load Balancer. Backend microservices are deployed as containerized services on ECS Fargate within private subnets, ensuring scalability and isolation. Data persistence and optimization are handled using Amazon Aurora, Redis caching, and OpenSearch, while property images are stored in Amazon S3. This design ensures high availability, security, and efficient traffic handling.

4.2 Presentation Layer

This layer represents the entry point for end users, including customers and agents. Users interact with the system through a frontend application delivered via **Amazon CloudFront**. **Amazon Route 53** resolves domain requests, while **Amazon Cognito** provides secure user authentication and role-based access control. This layer ensures secure and low-latency user access.

4.3 Application Layer

The core business logic resides in the application layer. Backend services are implemented as independent microservices and deployed as Docker containers on **AWS ECS Fargate** within private subnets. Customer-facing services handle read-heavy operations, while agent services manage write-intensive operations. This separation enables independent scaling and fault isolation.

4.4. Data Management Layer

This layer handles persistent storage and data optimization. **Amazon Aurora (RDS)** provides relational data storage with separate reader and writer endpoints. **Amazon ElastiCache (Redis)** is used for caching frequently accessed data, and **Amazon OpenSearch** enables fast and scalable search functionality. Property images and static assets are stored in **Amazon S3**.

AWS Services Used

AWS Service Configuration

5.1 PM2

PM2 is used in EstateCore to run and manage the Node.js backend services on the server. It ensures the application remains online by automatically restarting services in case of failure.

5.2 MongoDB Atlas

MongoDB Atlas is used to store EstateCore's application data such as user bookings, property listings, and agent information. Being a managed database, it provides scalability, security, and high availability.

5.3 Amazon EC2

Amazon EC2 is used to host the EstateCore backend services. It provides a virtual server environment where the application and PM2-managed services are deployed.

5.4 Amazon VPC

Amazon VPC is used to isolate EstateCore's cloud infrastructure. It ensures secure networking by placing backend services inside controlled virtual networks.

5.5 Amazon S3

Amazon S3 is used to store property images uploaded by agents. This allows scalable and durable storage without overloading the application server.

5.6 Security Groups (SG)

Security Groups are used to control network access to EstateCore resources. They allow only authorized traffic to reach EC2 instances and databases.

5.7 Elastic IP

Elastic IP was used to assign a static public IP address to the EC2 instance hosting the EstateCore backend. This ensures consistent access to the backend API even after instance restarts.

5.8 Git Bash

Git Bash was used as the command-line interface on the local machine to manage version control operations. It was also used to securely transfer code and interact with the EC2 server via SSH and SCP commands.

5.9 Cloudinary

Cloudinary is used for cloud-based image storage and optimization in the EstateCore project. Property images uploaded by users are stored on Cloudinary, which reduces server load and ensures fast image delivery through optimized URLs.

Implementation Details

Step 1: AWS Academy Learner Lab Initialization

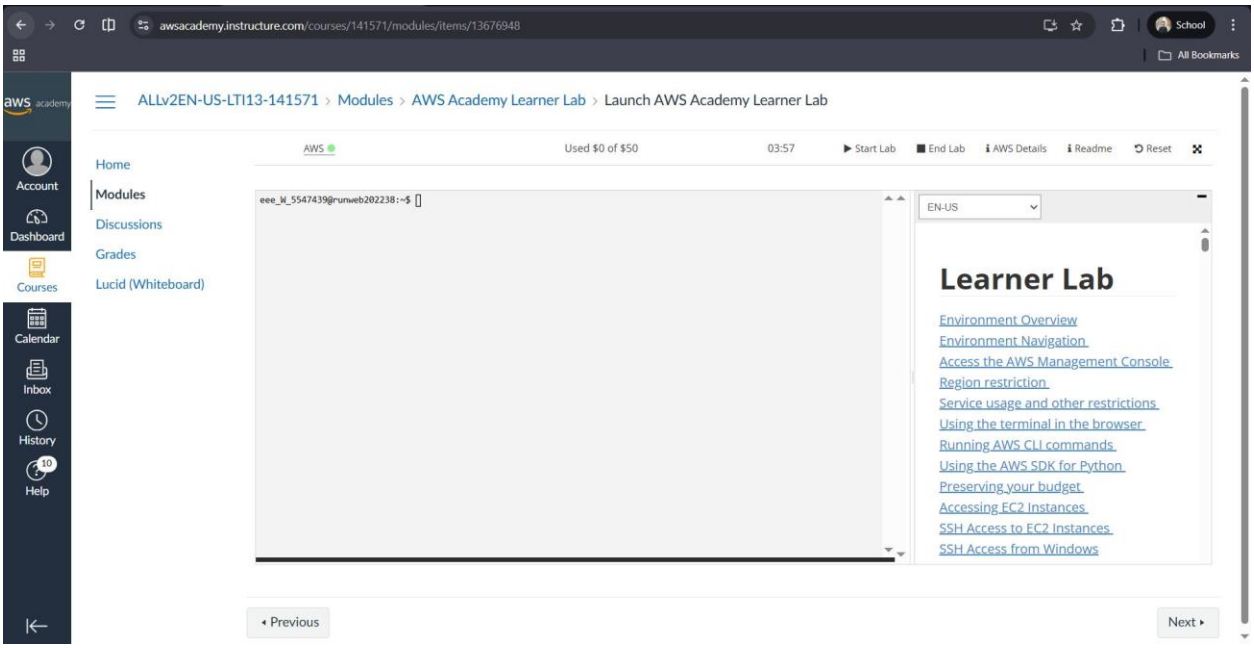

Figure 3: AWS Academy Learner Lab Initialization

AWS Academy Learner Lab was launched to obtain a temporary cloud environment. This lab provides controlled AWS access for deploying and testing cloud resources without incurring real billing costs.

Step 2: Accessing AWS Management Console

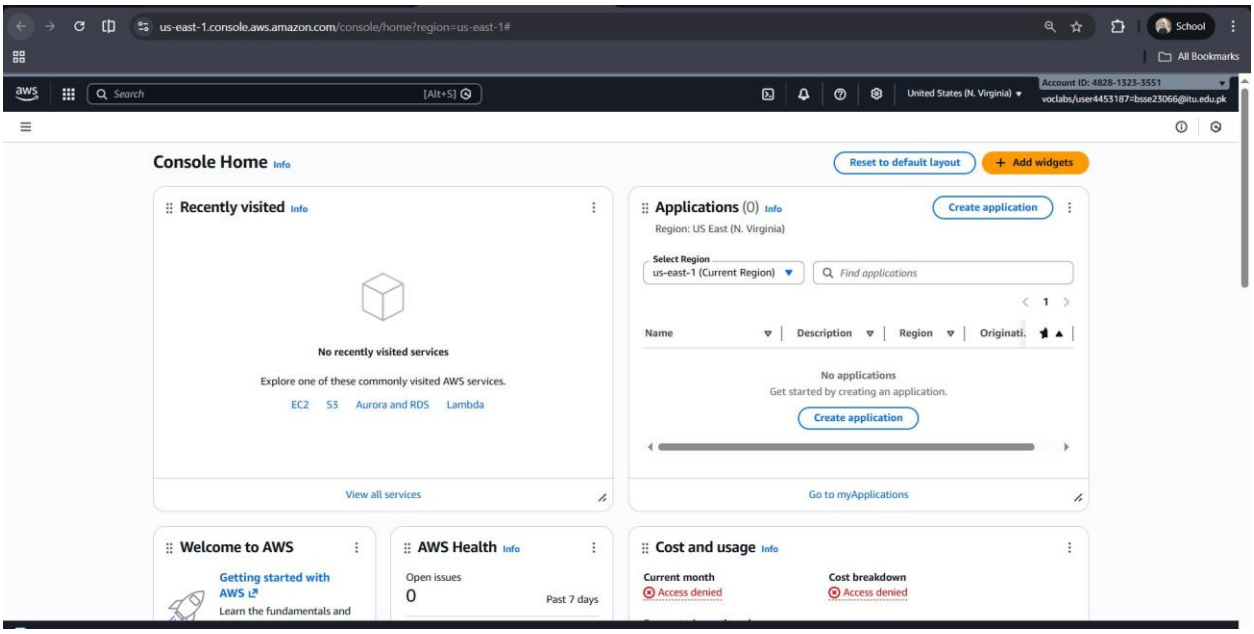

Figure 4: Accessing AWS Management Console

The AWS Management Console was accessed from the Learner Lab environment. This console serves as the central interface for creating and managing all AWS services used in the project.

Step 3: IAM Dashboard Configuration

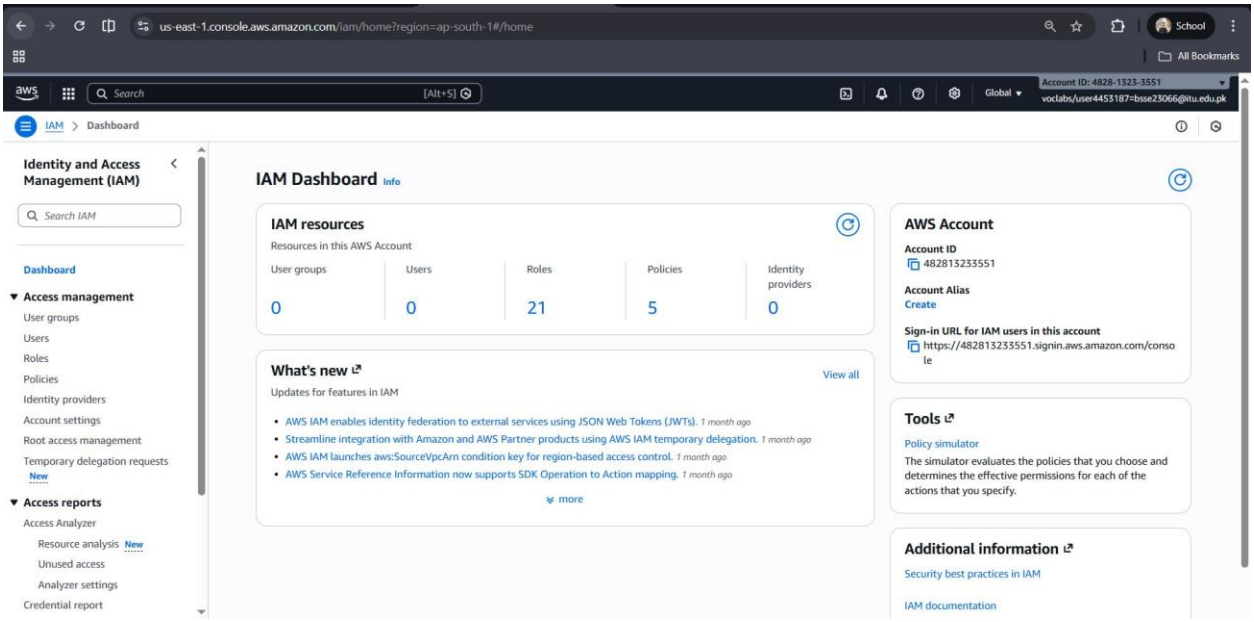

Figure 5: IAM Dashboard Configuration

The IAM dashboard was used to manage identities and permissions. It provides an overview of users, roles, and policies to ensure controlled and secure access to AWS resources.

Step 4: IAM User Creation

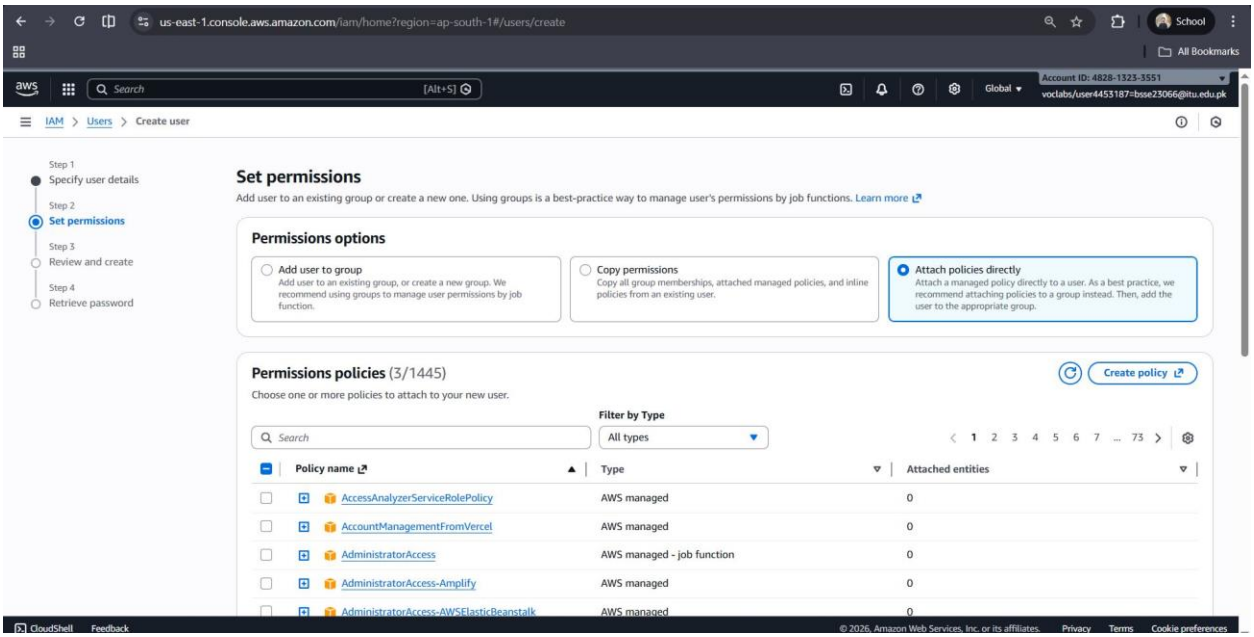

Figure 6: IAM User Creation

A dedicated IAM user was created for the project to avoid using the root account. This follows AWS security best practices by limiting privileges and improving access control.

Step 5: Assigning IAM Permissions

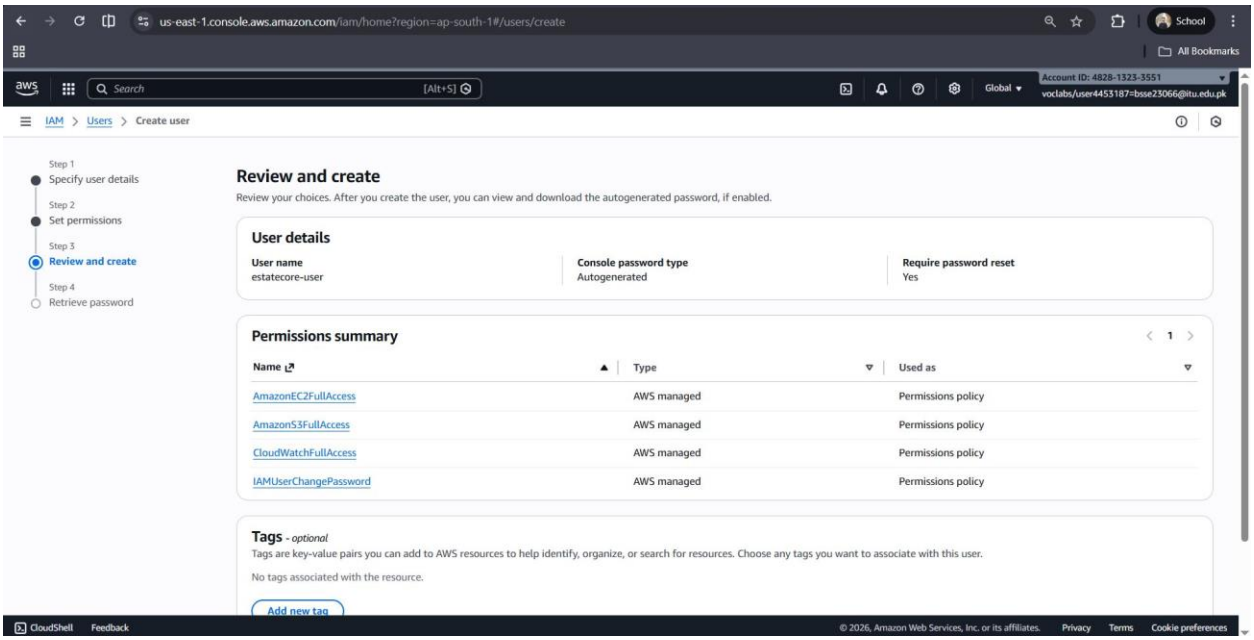

Figure 7: Assigning IAM Permissions

Required AWS-managed policies such as EC2 and S3 access were attached to the IAM user. These permissions allow the user to deploy infrastructure while maintaining controlled authorization.

Step 6: Launching EC2 Instance

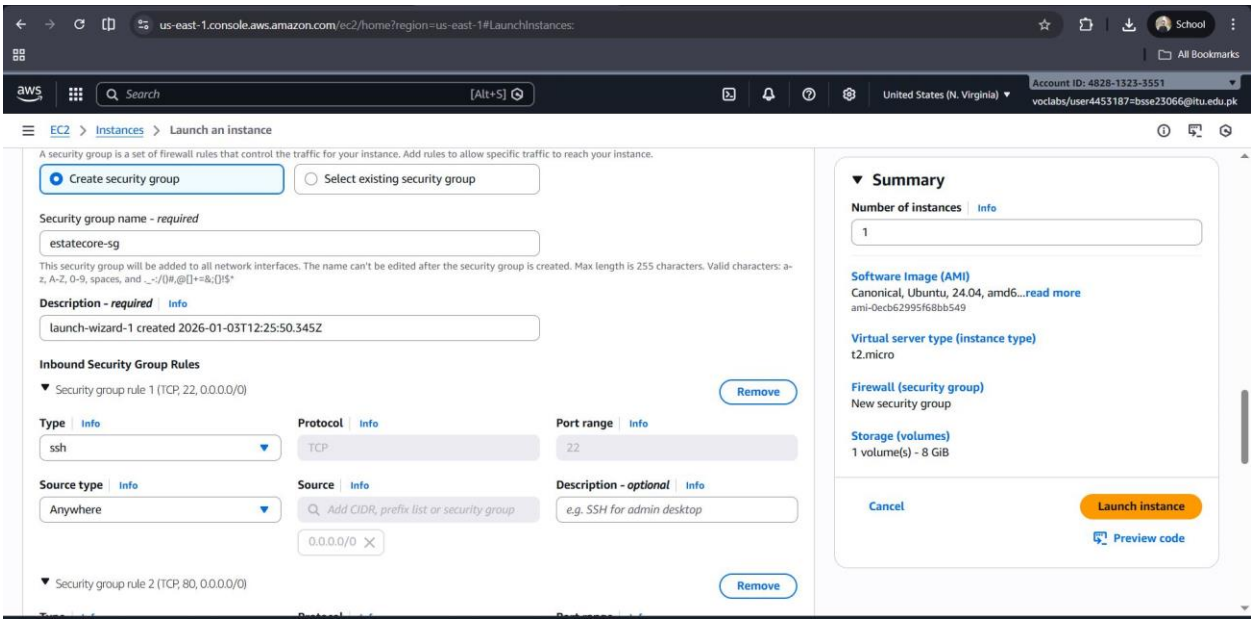

Figure 8: Launching EC2 Instance

An Amazon EC2 instance was launched using an Ubuntu AMI. This instance serves as the backend server for hosting the EstateCore application.

Step7: EC2 Instance Deployment

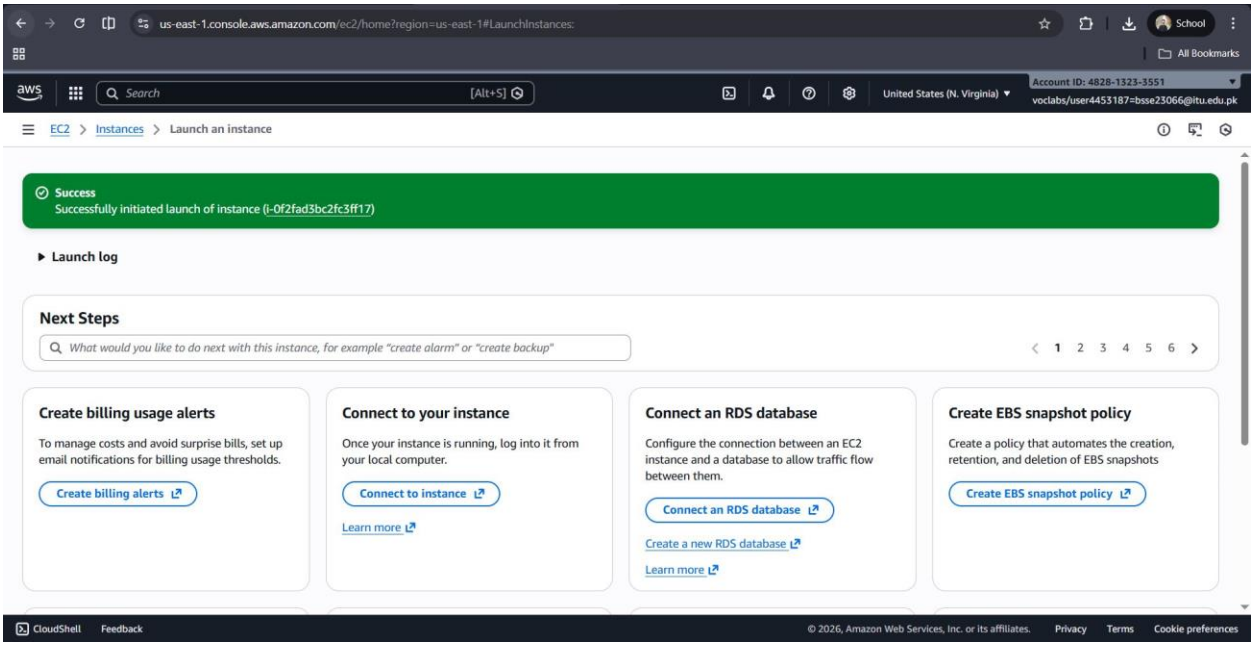

Figure 9: EC2 Instance Deployment

An Amazon EC2 instance was launched to host the backend services of the EstateCore application. This instance provides a dedicated virtual server environment for running the Node.js backend.

Step 8: Secure Server Access via SSH

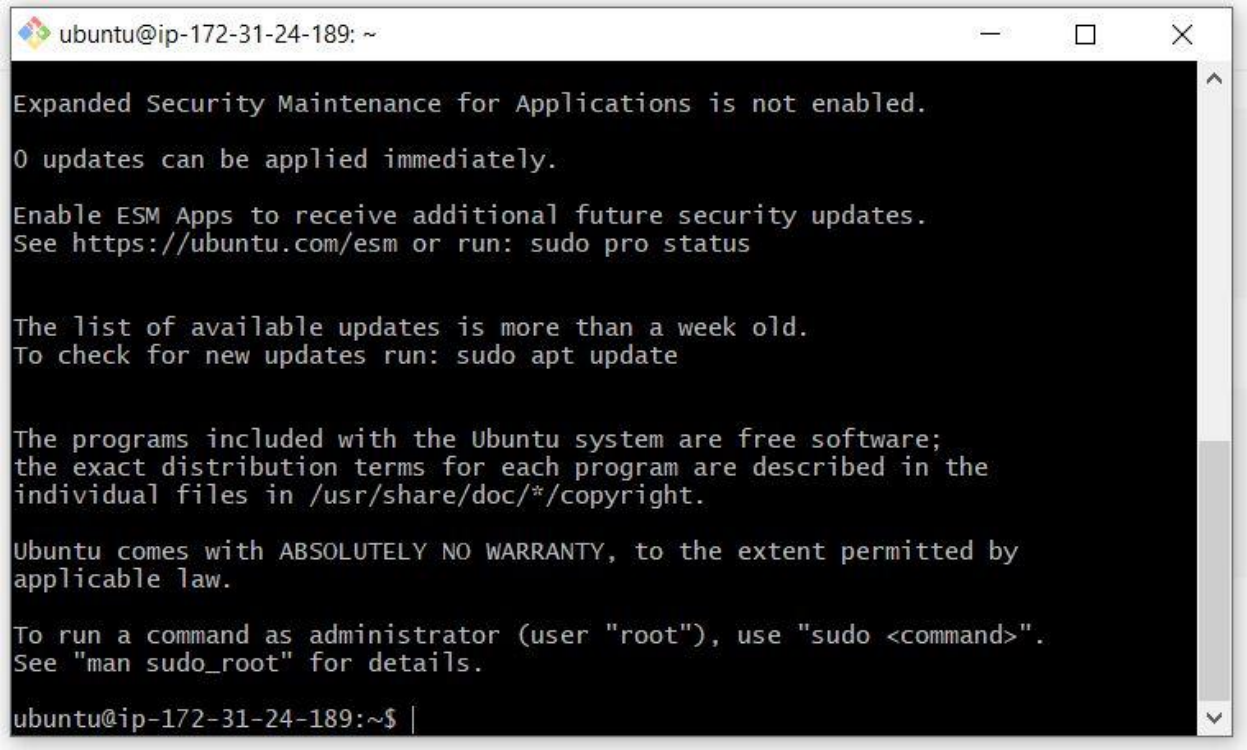

Figure 10: Secure Server Access via SSH

The EC2 instance was accessed securely using SSH. This allowed remote configuration, dependency installation, and application deployment on the Ubuntu server.

Step 9: Backend Code Transfer to Server

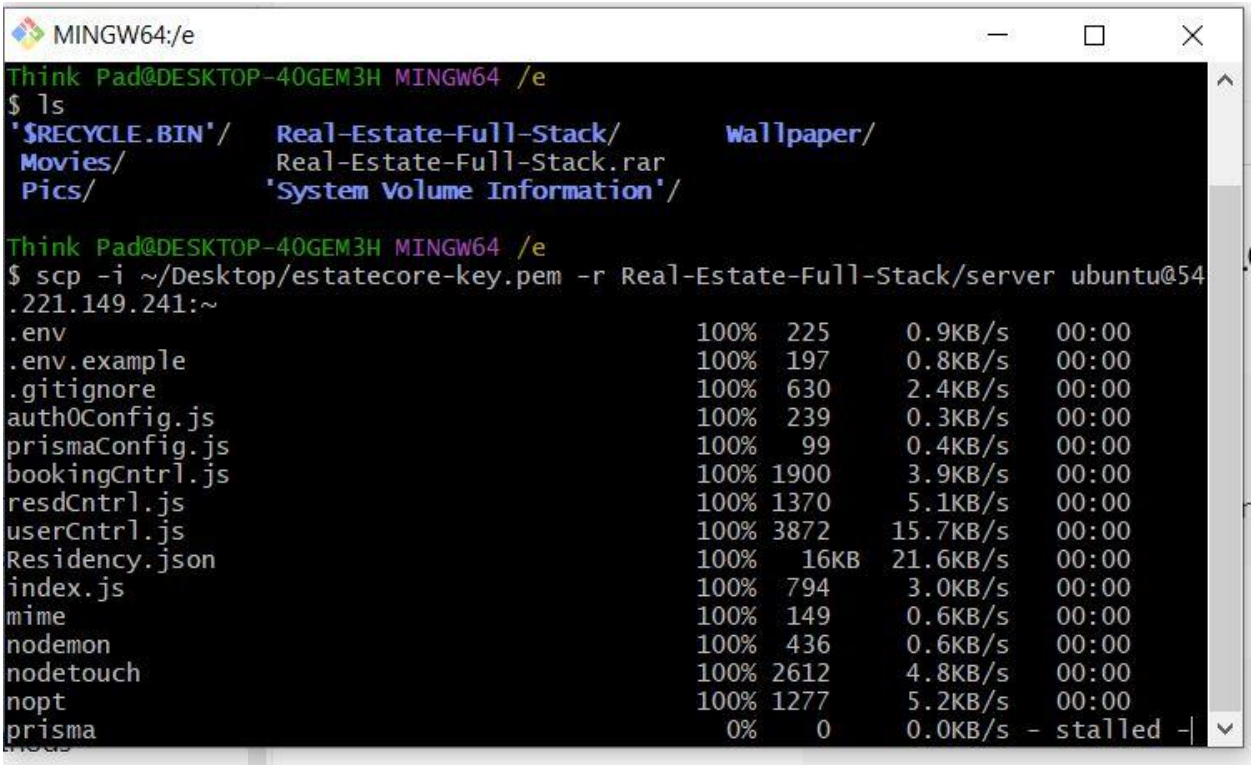

Figure 11: Backend Code Transfer to Server

The backend project files were transferred from the local machine to the EC2 instance using secure copy (SCP). This step ensured the application source code was available on the deployment server.

Step 10: Server Environment Setup

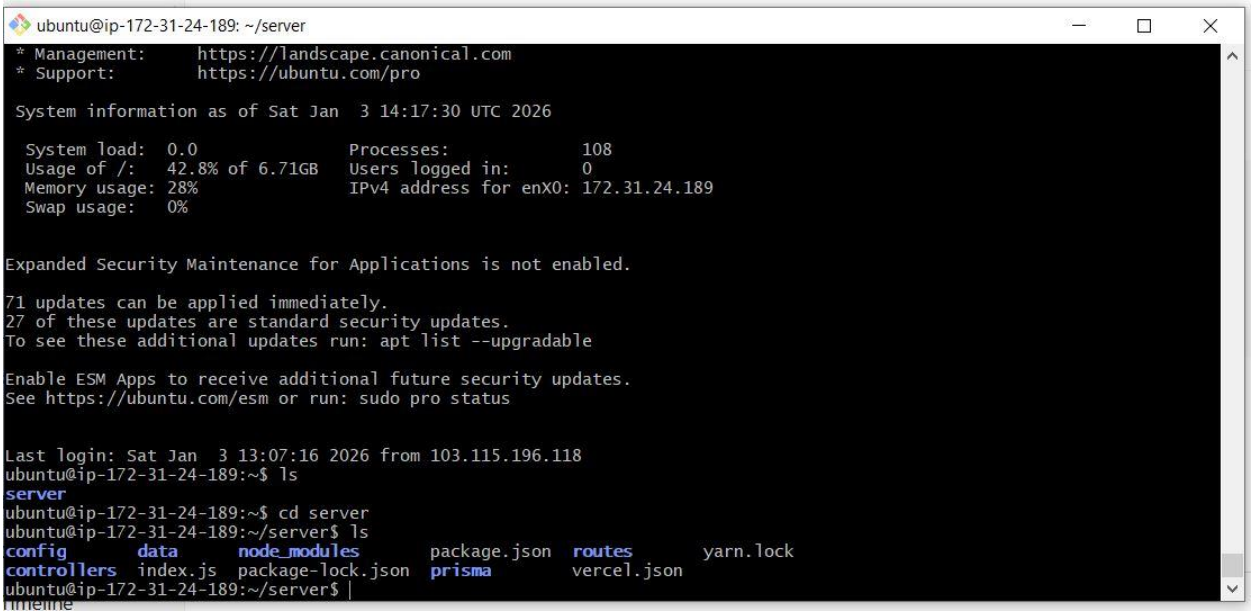

Figure 12: Server Environment Setup

Required dependencies and project files were verified on the EC2 instance. The backend directory structure was prepared for execution and process management.

Step 11: Backend API Verification

Figure 13: Backend API Verification

The backend API was tested through the public IP address of the EC2 instance to confirm successful deployment and server connectivity.

Step 12: Application Process Management using PM2

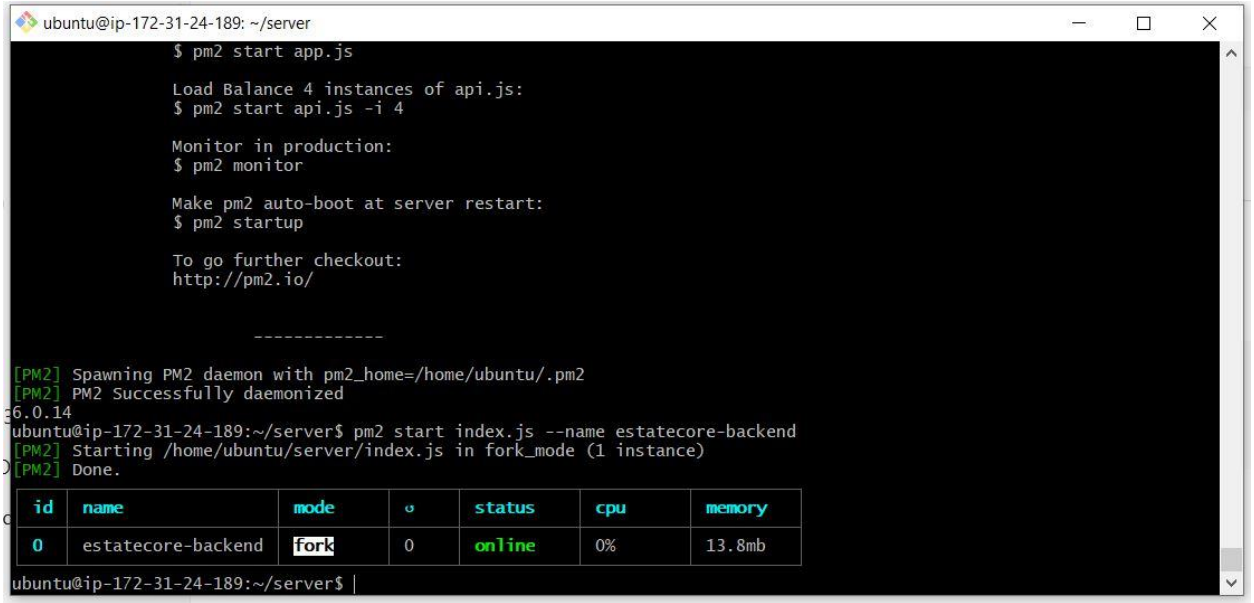

Figure 14: Application Process Management using PM2

PM2 was used to start and manage the Node.js backend service. It ensures the application runs continuously, automatically restarts on failure, and remains active after server reboots.

Step 13: S3 Bucket Creation

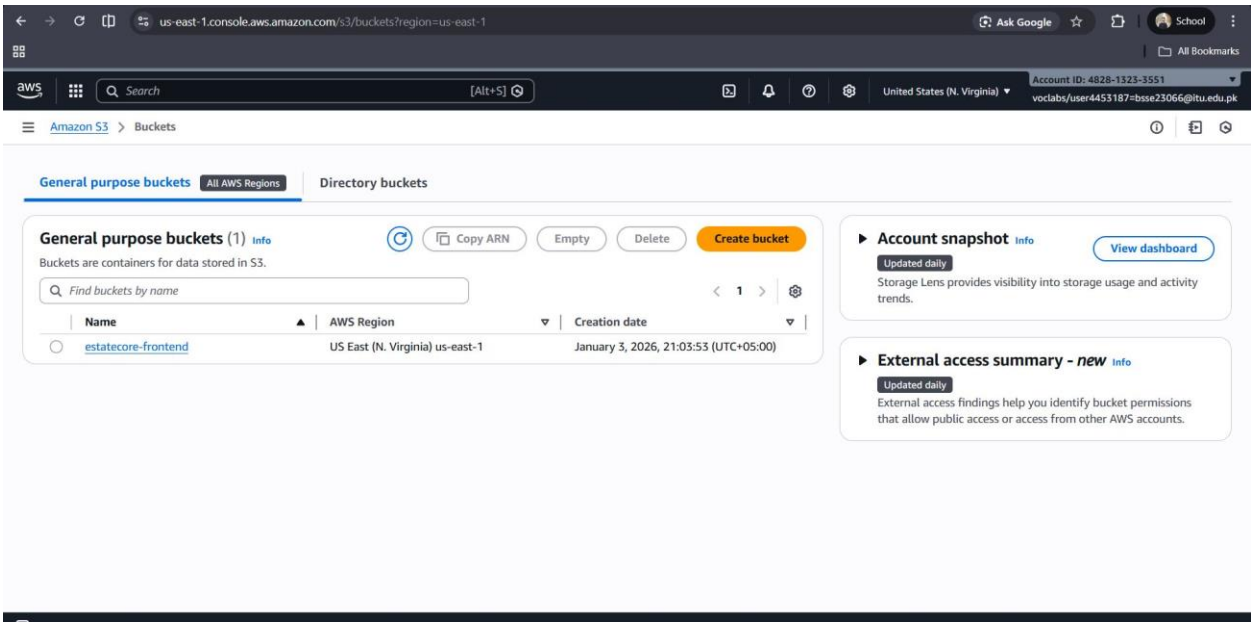

Figure 15: S3 Bucket Creation

An Amazon S3 bucket was created to store frontend assets and property images. This enables scalable, durable, and cost-effective storage separate from the application server.

Step 14: S3 Bucket Policy Configuration

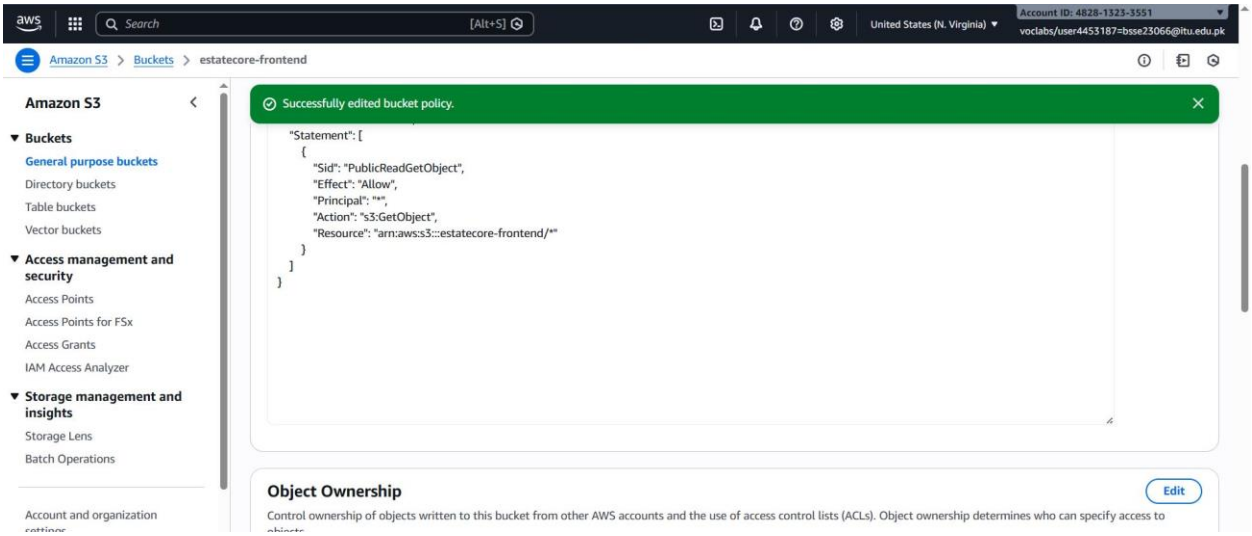

Figure 16: S3 Bucket Policy Configuration

A public-read bucket policy is applied to allow users to access frontend files over the internet.

Step 15: Frontend Deployment on Amazon S3

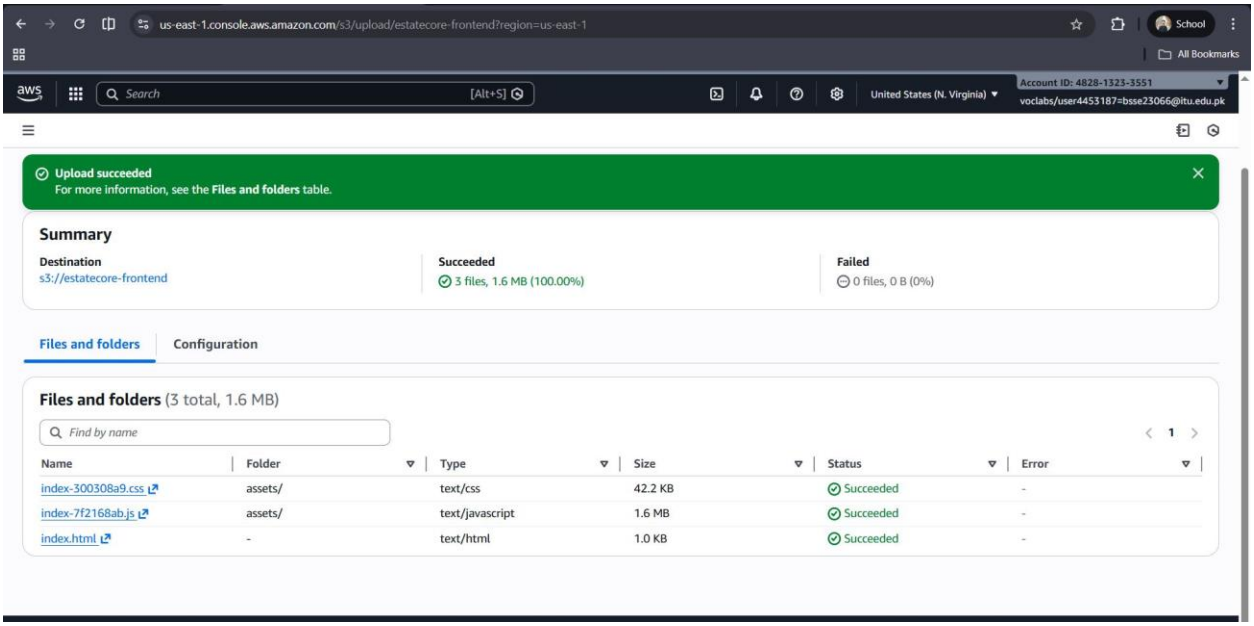

Figure 17: Frontend Deployment on Amazon S3

The frontend build files of the EstateCore application were uploaded to an Amazon S3 bucket. This enabled static hosting of the frontend, allowing users to access the application through a scalable and highly available storage service.

Link: http://estatecore-frontend.s3-website-us-east-1.amazonaws.com/

Step 16: MongoDB Atlas Database View

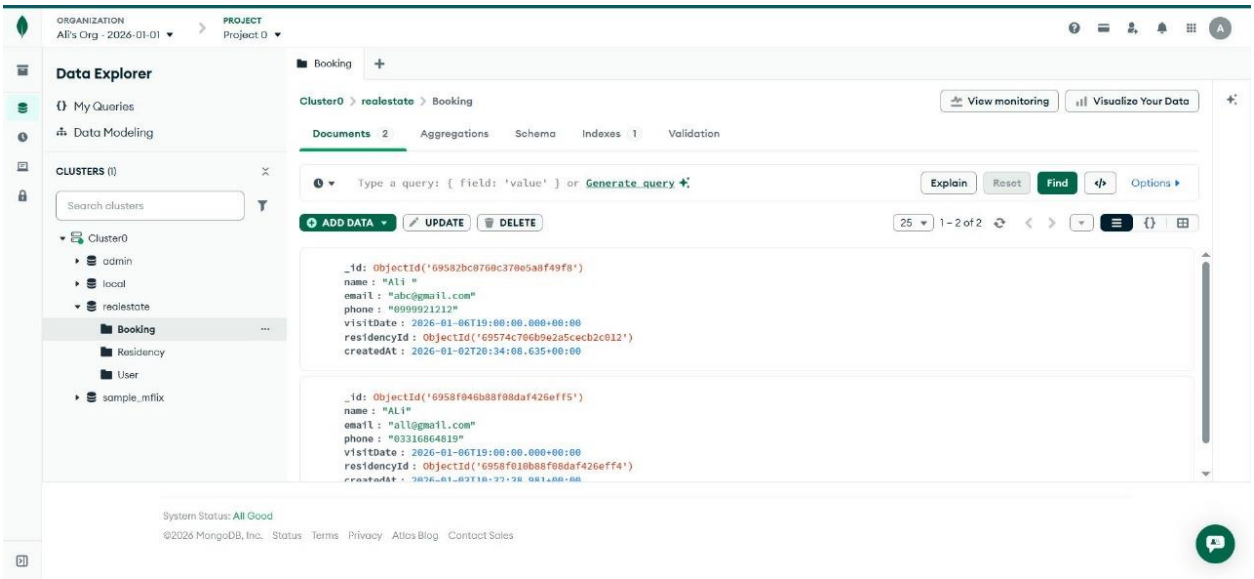

Figure 18: MongoDB Atlas Database View

This screen displays stored booking records in MongoDB Atlas, confirming database connectivity.

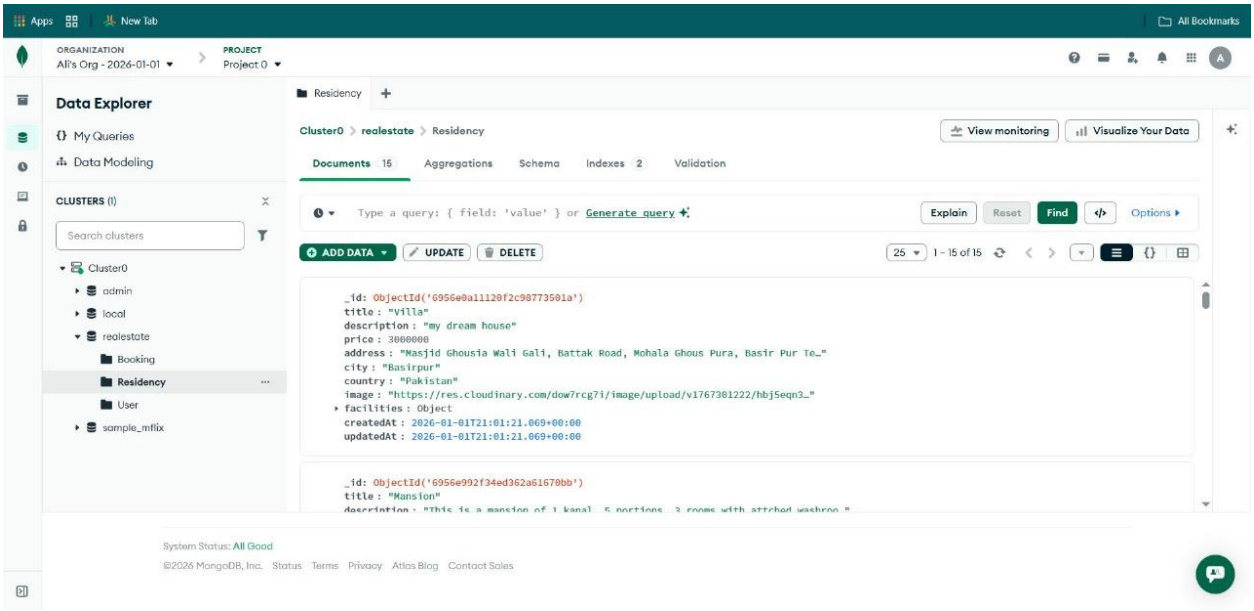

Figure 19: Another MongoDB Atlas Database View

This screen displays stored booking records in MongoDB Atlas, confirming database connectivity.

Step 17: S3 Static Website Hosting Settings

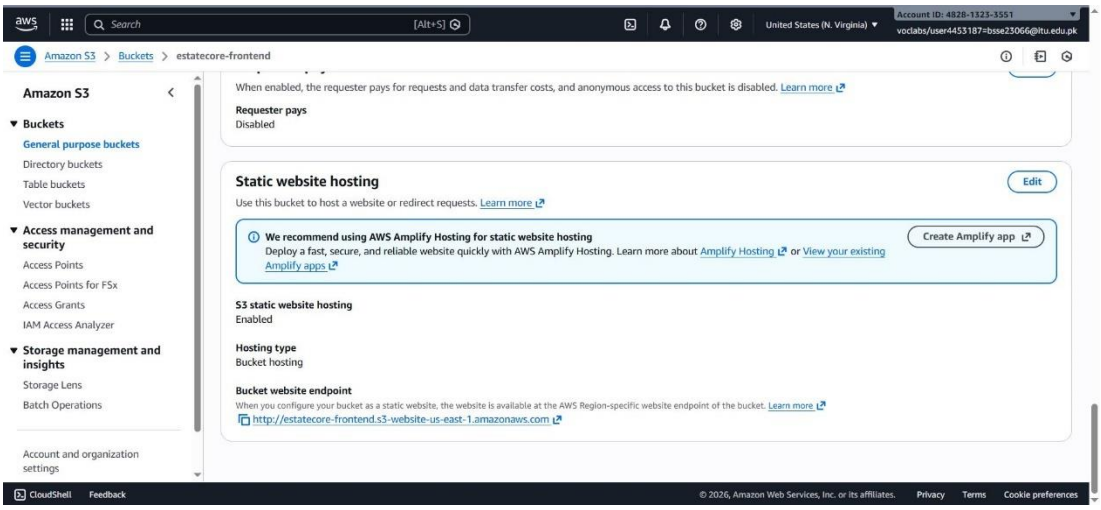

Figure 20: S3 Static Website Hosting Settings

Static website hosting is enabled here, generating a public URL for frontend access.

Security

7.1 Network Security

EstateCore provides network-level security by putting backend services on an Amazon VPC. Security Groups function as virtual firewalls, restricting inbound and outbound traffic to only permitted ports and IP addresses. This separation prevents the public from directly accessing crucial system components like application servers and databases.

7.2 Application Security

User access to the system is controlled through secure authentication mechanisms, ensuring that only authorized users can perform specific actions. Role-based access control differentiates customer and agent permissions, preventing unauthorized data modifications and protecting sensitive operations.

7.3 Data Security

Sensitive application data is securely stored in MongoDB Atlas, which provides encryption at rest and in transit. Property images stored in Amazon S3 are protected through controlled access policies, ensuring data integrity and confidentiality.

Conclusion

This project presented EstateCore, a cloud-based real-estate platform designed to address the scalability, performance, and availability limitations of traditional systems. By leveraging AWS services such as EC2, S3, VPC, and MongoDB Atlas, the system achieves secure deployment, efficient data management, and reliable backend execution using PM2. The implementation demonstrates the practical application of cloud computing and modern software engineering principles in building a production-ready system.

Future Work

Future enhancements may include implementing advanced security features such as HTTPS and role-based access policies, and adding real-time features like chat and notifications. Additionally, the system can be extended with analytics dashboards, AI-based property recommendations, and automated scaling mechanisms to further improve usability and system efficiency.

References

[1] Amazon Web Services, “Amazon EC2 User Guide,” AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/ec2/

[2] Amazon Web Services, “Amazon S3 User Guide,” AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/s3/

[3] Amazon Web Services, “Amazon VPC User Guide,” AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/vpc/

[4] Amazon Web Services, “Elastic IP Addresses,” AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

[5] PM2, “PM2 – Advanced Node.js Process Manager,” PM2 Documentation. [Online]. Available: https://pm2.keymetrics.io/